

Fast Dynamic Graph Algorithms for Parameterized Problems*

Yoichi Iwata[†]

Keigo Oka[‡]

Abstract

Fully dynamic graph is a data structure that (1) supports edge insertions and deletions and (2) answers problem specific queries. The time complexity of (1) and (2) are referred to as the update time and the query time respectively. There are many researches on dynamic graphs whose update time and query time are $o(|G|)$, that is, sublinear in the graph size. However, almost all such researches are for problems in P. In this paper, we investigate dynamic graphs for NP-hard problems exploiting the notion of fixed parameter tractability (FPT).

We give dynamic graphs for **Vertex Cover** and **Cluster Vertex Deletion** parameterized by the solution size k . These dynamic graphs achieve almost the best possible update time $O(\text{poly}(k) \log n)$ and the query time $O(f(\text{poly}(k), k))$, where $f(n, k)$ is the time complexity of any static graph algorithm for the problems. We obtain these results by dynamically maintaining an approximate solution which can be used to construct a small problem kernel. Exploiting the dynamic graph for **Cluster Vertex Deletion**, as a corollary, we obtain a quasilinear-time (polynomial) kernelization algorithm for **Cluster Vertex Deletion**. Until now, only quadratic time kernelization algorithms are known for this problem.

We also give a dynamic graph for **Chromatic Number** parameterized by the solution size of **Cluster Vertex Deletion**, and a dynamic graph for bounded-degree **Feedback Vertex Set** parameterized by the solution size. Assuming the parameter is a constant, each dynamic graph can be updated in $O(\log n)$ time and can compute a solution in $O(1)$ time. These results are obtained by another approach.

1 Introduction

1.1 Background

1.1.1 Parameterized Algorithms

Assuming $P \neq NP$, there are no polynomial-time algorithms solving NP-hard problems. On the other hand, some problems are efficiently solvable when a certain parameter, e.g. the size of a solution, is small. *Fixed parameter tractability* is one of the ways to capture such a phenomenon.

A problem is in the class *fixed parameter tractable (FPT)* with respect to a parameter k if there is an algorithm that solves any problem instance of size n with parameter k in $O(n^d f(k))$ time (*FPT time*), where d is a constant and f is some computable function.

*A preliminary version of this paper appears in the proceedings of SWAT 2014.

[†]Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo. y.iwata@is.s.u-tokyo.ac.jp

[‡]Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo. ogiekako@is.s.u-tokyo.ac.jp

Problem	Update Time	Query Time	Section
Vertex Cover	$O(k^2)$	$f_{VC}(k^2, k)$	3
Cluster Vertex Deletion	$O(k^8 + k^2 \log n)$	$f_{CVD}(k^5, k)$	4
Cluster Vertex Deletion	$O(8^k k^6)$	$O(1)$	5
Chromatic Number	$O(2^{2^k} \log n)^1$	$O(1)$	6
Feedback Vertex Set	$O(7.66^k k^3 + 2^k k^3 d^3 \log n)$	$O(1)$	7

Table 1: The time complexities of the dynamic graphs in this paper. d is the degree bound, and n is the number of the vertices. The parameter for **Chromatic Number** is cvd number (the size of a minimum cluster vertex deletion), and parameters for the other problems are its solution size.

1.1.2 Dynamic Graphs

(Fully) dynamic graph is a data structure that supports edge insertions, edge deletions, and answers certain problem specific queries. There are a lot of theoretical research on dynamic graphs for problems that belong to P, such as **Connectivity** [15, 16, 30, 33, 11, 19], k -**Connectivity** [16, 11], **Minimum Spanning Forest** [16, 11], **Bipartiteness** [16, 11], **Planarity Testing** [18, 11, 20], **All-pairs Shortest Path** [31, 6, 32, 27, 25] and **Directed Connectivity** [5, 23, 24, 26, 28], and races for faster algorithms are going on.

On the contrary there have been few research on dynamic graphs related to FPT algorithms. To the best of our knowledge, a dynamic data structure for counting subgraphs in sparse graphs proposed by Zdeněk Dvořák and Vojtěch Tůma [10] and a dynamic data structure for tree-depth decomposition proposed by Zdeněk Dvořák, Martin Kupec and Vojtěch Tůma [9] are only such dynamic graphs. Both data structures support insertions and deletions of edges, and compute the solution of the problems in time depending only on k , where k is the parameter of the problem. For a fixed property expressed in monadic second-order logic, the dynamic graph in [9] also can answer whether the current graph has the property. For both algorithms, hidden constants are (huge) exponential in k . In particular, update time of both algorithms become super-linear in graph size n even if k is very small, say $O(\log \log n)$.

1.2 Our Contribution

In this paper, we investigate dynamic data structures for basic graph problems in FPT. Table 1 shows the problems we deal with and the time complexities of the algorithms.

1.2.1 Dynamic Graph for Vertex Cover and Cluster Vertex Deletion

In Section 3 and 4, we present fully dynamic graphs for **Vertex Cover** and **Cluster Vertex Deletion**, respectively. Both dynamic data structures support additions and deletions of edges, and can answer the solution of the problem in time depending only on the solution size k .

For the dynamic graph for **Vertex Cover**, the time complexity of an edge addition or deletion is $O(k^2)$ and the one of a query is $f_{VC}(k^2, k)$ where $f_{VC}(n, k)$ is the time complexity of any static algorithm for **Vertex Cover** on a graph of size n .

¹More precisely, it is $O(B_{2k}(4^k k^3 + \log n))$ as proved in Section 6. (B_n is the *Bell number* for n , the number of ways to divide a set with n elements.)

For the dynamic graph for **Cluster Vertex Deletion**, the time complexity of an update is $O(k^8 \log n)$ and the one of a query is $f_{CVD}(k^5, k)$ where $f_{CVD}(n, k)$ is the time complexity of any static algorithm for **Cluster Vertex Deletion** on a graph of size n . The extra $\log n$ factor arises because we use persistent data structures to represent some vertex sets. This enables us to copy a set in constant time.

Note that the time complexity of an update is $\text{poly}(k)\text{polylog}(n)$ for both algorithms, instead of an exponential function in k . As for the time complexity of a query, its exponential term in k is no more than any static algorithms.

Let us briefly explain how the algorithms work. Throughout the algorithm, we keep an approximate solution. When the graph is updated, we efficiently construct a $\text{poly}(k)$ size kernel by exploiting the approximate solution, and then compute a new approximate solution on this kernel. Here, we compute not an exact solution but an approximate solution to achieve the update time polynomial in k . To answer a query, we apply a static exact algorithm to the kernel.

To see goodness of these algorithms, consider the situation such that a query is applied for every r updates. A trivial algorithm answers a query by running a static algorithm. Let the time complexity of the static algorithm be $O(f(n, k))$. In this situation, to deal with consecutive r updates and one query, our algorithm takes $O(r\text{poly}(k)\text{polylog}(n) + f(\text{poly}(k), k))$ time, and the trivial algorithm takes $O(f(n, k))$ time. For example, let $f(n, k) = c^k + kn$ be the time complexity of the static algorithm. (The time complexity of the current best FPT algorithm for **Vertex Cover** is $O(1.2738^k + kn)$ [4].) Then if $r = \sqrt{n}$ and $c^k = \sqrt{n}$, the time complexity for the dynamic graph algorithm is $\sqrt{n}\text{polylog}(n) = o(n)$, sublinear in n . That is, our algorithm works well even if the number of queries is fewer than the number of updates. This is an advantage of the polynomial-time update. If $r = 1$, our algorithm is faster than the trivial algorithm whenever $c^k < n$. Even if c^k is the dominant term, our algorithm is never slower than the trivial algorithm.

Let us consider the relation between our results and the result by Dvořák, Kupec and Tůma [9]. The size of a solution of **Vertex Cover** is called *vertex cover number*, and the size of a solution of **Cluster Vertex Deletion** is called *cluster vertex deletion number* (*cvd number*). It is easy to show that tree-depth can be arbitrarily large even if cvd number is fixed and vice versa. Thus our result for **Cluster Vertex Deletion** is not included in their result. On the other hand, tree-depth is bounded by vertex cover number + 1. Thus their result indicates that **Vertex Cover** can be dynamically computed in $O(1)$ time if vertex cover number is a constant. However, if it is not a constant, say $O(\log \log n)$, the time complexity of their algorithm becomes no longer sublinear in n . The time complexity of our algorithm for **Vertex Cover** is further moderate as noted above.

As an application of the dynamic graph for **Cluster Vertex Deletion**, we can obtain a quasilinear-time kernelization algorithm for **Cluster Vertex Deletion**. To compute a problem kernel of a graph $G = (V, E)$, starting from an empty graph, we iteratively add the edges one by one while updating an approximate solution. Finally, we compute a kernel from the approximate solution. As shown in Section 4, the size of the problem kernel is $O(k^5)$ and the time for an update is $O(k^8 \log |V|)$. Thus, we obtain a polynomial kernel in $O(k^8 |E| \log |V|)$ time.

Protti, Silva and Szwarcfiter [21] proposed a linear-time kernelization algorithm for **Cluster Editing** applying modular decomposition techniques. On the other hand, to the best of our knowledge, for **Cluster Vertex Deletion**, only quadratic time kernelization algorithms [17] are known (until now). Though **Cluster Vertex Deletion** and **Cluster Editing** are similar problems, it seems that their techniques cannot be directly applied to obtain a linear-time kernelization algorithm for **Cluster Vertex Deletion**.

1.2.2 Dynamic Graph for Chromatic Number Parameterized by CVD Number

The study of problems parameterized by cvd number was initiated by Martin Doucha and Jan Kratochvíl [7]. They studied the fixed parameter tractability of basic graph problems related to coloring and Hamiltonicity, and proved that the problems **Equitable Coloring**, **Chromatic Number**, **Hamiltonian Path** and **Hamiltonian Cycle** are in FPT parameterized by cvd number.

In this paper, we also obtained a fully dynamic data structure for **Chromatic Number** parameterized by cvd number. Assuming the cvd number is a constant, the time complexity of an update and a query is $O(1)$. In our algorithm, we maintain not only a minimum cluster vertex deletion but also more detailed information, equivalent vertex classes in each cluster. We consider two vertices in a same cluster are equivalent if their neighbors in the current solution are exactly same. To update such underlying data structures efficiently, we present another dynamic graph for **Cluster Vertex Deletion** in Section 5. Unlike the algorithms in section 3 and 4, this algorithm deals with an update in exponential time in k , and it seems difficult to make it polynomial by the need to maintain equivalent classes.

Then, we design a dynamic graph for **Chromatic Number** in Section 6. In the algorithm, for each update we consider each possible coloring for the current minimum cluster vertex deletion and compute the minimum number of colors to color the other vertices exploiting the equivalent classes and a flow algorithm. Our algorithm is based on a static algorithm in [14].

1.2.3 Dynamic Graph for Bounded-Degree Feedback Vertex Set

Finally we present a fully dynamic data structure for bounded-degree **Feedback Vertex Set** in Section 7. Despite the restriction of the degree, we believe the result is still worth mentioning because the algorithm is never obvious. The algorithm is obtained by exploiting a theorem in [14] and a classic Link-Cut Tree data structure introduced by Sleator and Tarjan [29]. As with [14], we use the idea of *iterative compression*. Iterative compression is the technique introduced by Reed, Smith and Vetta [22]. Its central idea is to iteratively compute a minimum solution with size k making use of a solution with size $k + 1$.

This algorithm also takes exponential time in k for an update mainly because we consider all $O(2^k)$ possibility of $X \cap X'$ where X is the current solution and X' is the updated solution.

It seems an interesting open question whether it is possible to construct an efficient dynamic graph without the degree restriction.

2 Notations

Let $G = (V, E)$ be a simple undirected graph with vertices V and edges E . We consider that each edge in E is a set of vertices of size two. Let $|G|$ denote the *size* of the graph $|V| + |E|$. The *neighborhood* $N_G(v)$ of a vertex v is $\{u \in V \mid \{u, v\} \in E\}$, and the neighborhood $N_G(S)$ of a vertex set $S \subseteq V$ is $\bigcup_{v \in S} N_G(v) \setminus S$. The *closed neighborhood* $N_G[v]$ of a vertex v is $N_G(v) \cup \{v\}$, and the closed neighborhood $N_G[S]$ of a vertex set $S \subseteq V$ is $N_G(S) \cup S$. Let the *incident edges* $\delta_G(v)$ of a vertex v be the set of edges incident to the vertex v . The *cut edges* $\delta_G(S, T)$ between two disjoint vertex subsets S and T are $\{\{u, v\} \in E \mid u \in S, v \in T\}$. We denote the degree of a vertex v by $d_G(v)$. We omit the subscript if the graph is apparent from the context. The *induced subgraph* $G[S]$ of a vertex set S is the graph $(S, \{e \in E \mid e \subseteq S\})$. For an edge subset $F \subseteq E$, let $G - F$ be the graph $(V, E \setminus F)$.

Algorithm 1 compute a 2-approximate solution

```
1:  $X_0 := \emptyset$ 
2:  $V' := \emptyset$ 
3: for all  $x$  in  $X$  do
4:   if  $d(x) > |X|$  then  $X_0 := X_0 \cup \{x\}$ 
5:   else  $V' := V' \cup N[x]$ 
6:  $V' := V' \setminus X_0$ 
7:  $Y :=$  2-approximate solution for Vertex Cover of  $G[V']$ 
8:  $X' := X_0 \cup Y$ 
```

By default, we use $k(G)$ or k to denote the parameter value of the current graph G . When an algorithm updates a graph G to G' , we use $k = \max\{k(G), k(G')\}$ as a parameter. Note that, $k(G)$ and $k(G')$ are not greatly different in most problems. In particular, it is easy to prove that for all problems we deal with in this paper, $k(G')$ is at most $k(G) + 1$.

3 Dynamic Graph for Vertex Cover

Let $G = (V, E)$ be a graph. **Vertex Cover** is the problem of finding a minimum set of vertices that covers all edges. Let $k = k(G)$ be the size of a minimum vertex cover of G . The current known FPT algorithm solving **Vertex Cover** whose exponential function in k is smallest is by Chen, Kanj and Xia [4], and its running time is $O(|G|k + 1.2738^k)$. Let us now state the main result of this section.

Theorem 1. *There is a data structure representing a graph G which supports the following three operations.*

1. *Answers the solution for **Vertex Cover** of the current graph G .*
2. *Add an edge to G .*
3. *Remove an edge from G .*

*Let k be the size of a minimum vertex cover of G . Then the time complexity of an edge addition or removal is $O(k^2)$, and of a query is $O(f(k^2, k))$, where $f(|G|, k)$ is the time complexity of any static algorithm for **Vertex Cover** on a graph of size $|G|$.*

Note that the update time is polynomial in k , and the exponential term in k of the query time is same to the one of the static algorithm.

Our dynamic data structure is simply represented as a pair of the graph $G = (V, E)$ itself and a 2-approximate solution $X \subseteq V$ for **Vertex Cover** of G , that is, we maintain a vertex set X such that X is a vertex cover of G and $|X| \leq 2k(G)$.

For both query and update, we compute a problem kernel. To do this, we exploit the fact that we already know rather small vertex cover X . When an edge $\{u, v\}$ is added to G , we add u to X making X a vertex cover and use Algorithm 1 to compute a new 2-approximate solution X' of G . When an edge is removed from G , we also use Algorithm 1 to compute a new 2-approximate solution.

Lemma 1. *Algorithm 1 computes a 2-approximate solution X' in $O(k^2)$ time, where $k = k(G)$.*

Proof. Let X^* be a minimum vertex cover of the updated graph. We have $|X^*| \leq |X|$. If $x \notin X^*$ for some vertex $x \in X_0$, $N(x)$ must be contained in X^* . Thus it holds that $|X^*| \geq |N(x)| = d(x) > |X|$, which is a contradiction. Therefore, it holds that $X_0 \subseteq X^*$. At line 7 of Algorithm 1, V' equals to $N[X \setminus X_0] \setminus X_0$. Thus we have:

- (1) $X^* \setminus X_0$ is a vertex cover of $G[V']$ because $X_0 \cap V' = \emptyset$ and X^* is a vertex cover of G , and
- (2) any vertex cover of $G[V']$ together with X_0 covers all edges in G because all edges not in $G[V']$ are covered by X_0 .

Putting (1) and (2) together, we can prove that $X^* \setminus X_0$ is a minimum vertex cover of $G[V']$.

Since Y is a 2-approximate solution on $G[V']$ and $X^* \setminus X_0$ is a minimum vertex cover of $G[V']$, we have $|Y| \leq 2|X^* \setminus X_0|$. From (2), $X' = X_0 \cup Y$ is a vertex cover of G . Thus X' is a 2-approximate solution because $|X'| = |X_0| + |Y| \leq |X_0| + 2|X^* \setminus X_0| \leq 2|X^*|$.

The size of X is at most $2k + 1$, and thus the size of V' at line 7 is $O(k^2)$. Moreover, the number of edges in $G[V']$ is $O(k^2)$ because for each edge in $G[V']$, at least one endpoint lies on $X \setminus X_0$ and the degree of any vertex x in $X \setminus X_0$ is at most $|X|$. A 2-approximate solution can be computed in linear time using a simple greedy algorithm [13]. Thus the total time complexity is $O(k^2)$. \square

To answer a query, we use almost the same algorithm as Algorithm 1, but compute an exact solution at line 7 instead of an approximate solution. The validity of the algorithm can be proved by almost the same argument. The bottleneck part is to compute an exact vertex cover of the graph $G[V']$. Since the size of the solution is at most k , we can obtain the solution in $O(f(k^2, k))$ time where $f(|G|, k(G))$ is the time complexity of any static algorithm for **Vertex Cover** on a graph of size $|G|$. For example, using the algorithm in [4], we can compute the solution in $O(k^3 + 1.2738^k)$ time. We have finished the proof of Theorem 1.

4 Dynamic Graph for Cluster Vertex Deletion

4.1 Problem Definition and Time Complexity

A graph is called *cluster graph* if every its connected component is a clique, or equivalently, it contains no *induced* path with three vertices (P_3). Each maximal clique in a cluster graph is called a *cluster*. Given a graph, a subset of its vertices is called a *cluster vertex deletion* if its removal makes the graph a cluster graph. **Cluster Vertex Deletion** is the problem to find a minimum cluster vertex deletion. We call the size of a minimum cluster vertex deletion as a *cluster vertex deletion number* or a *cvd number* in short.

There is a trivial algorithm to find a 3-approximate solution for **Cluster Vertex Deletion** with time complexity $O(|E||V|)$ [17]. The algorithm greedily finds P_3 and adds all the vertices on the path to the solution until we obtain a cluster graph. According to [17], it is still open whether it is possible to improve the trivial algorithm or not.

Let us now state the main result of this section.

Theorem 2. *There is a data structure representing a graph G which supports the following three operations.*

X	3-approximate solution
C_l for each cluster label l	the vertices in the cluster labeled l
l_u for each $u \in V \setminus X$	label of the cluster that u belongs to
L_x for each $x \in X$	$\{l_u \mid u \in N(x) \setminus X\}$
$P_{x,l}^+$ for each $x \in X$ and $l \in L_x$	$C_l \cap N(x)$
$P_{x,l}^-$ for each $x \in X$ and $l \in L_x$	$C_l \setminus N(x)$

Table 2: Variables maintained in the algorithm

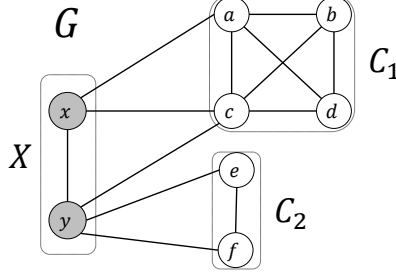


Figure 1: An example of a graph and a 3-approximate solution.

1. Answers the solution for Cluster Vertex Deletion of the current graph G .
2. Add an edge to G .
3. Remove an edge from G .

Let k be the cvd number of G . Then the time complexity of an edge addition or removal is $O(k^8 + k^2 \log |V|)$, and of a query is $O(f(k^5, k))$, where $f(|G|, k)$ is the time complexity of any static algorithm for Cluster Vertex Deletion on a graph of size $|G|$.

As the static algorithm, we can use an $O(2^k k^9 + |V||E|)$ -time algorithm by Hüffner, Komusiewicz, Moser, and Niedermeier [17] or an $O(1.9102^k |G|)$ -time algorithm by Boral, Cygan, Kociumaka, and Pilipczuk [2].

4.2 Data Structure

We dynamically maintain the information listed in Table 2. We always keep a 3-approximate solution X . Each cluster in $G[V \setminus X]$ is assigned a distinct *cluster label*. For each cluster label l , C_l is the set of vertices on the cluster having the label l . We keep the vertex set C_l by using a persistent data structure that supports an update in $O(\log |C_l|)$ time. One of such data structures is a persistent red-black tree developed by Driscoll, Sarnak, Sleator and Tarjan [8]. The reason why the persistent data structure is employed is that it enables us to copy the set in constant time. For each $u \in V \setminus X$, l_u is a label of the cluster u belongs to. For a vertex x and a cluster, we say that x is incident to the cluster if at least one vertex in the cluster is incident to x . For each $x \in X$, $L_x = \{l_u \mid u \in N(x) \setminus X\}$ is the labels of the clusters that x is incident to. For each $x \in X$ and $l \in L_x$, $P_{x,l}^+ = C_l \cap N(x)$ is the set of the neighbors of x in C_l and $P_{x,l}^- = C_l \setminus N(x)$ is the set of the

Algorithm 2 add $u \in V \setminus X$ to X

```
1:  $l := l_u$ 
2: remove  $u$  from  $C_l$ 
3: for all any  $x \in X$  such that  $l \in L_x$  do
4:   if  $\{x, u\} \in E$  then
5:     remove  $u$  from  $P_{x,l}^+$ 
6:     If  $P_{x,l}^+$  becomes empty, remove  $l$  from  $L_x$ 
7:   else
8:     remove  $u$  from  $P_{x,l}^-$ 
9: add  $u$  to  $X$ 
10: if  $C_l$  is still not empty then
11:    $L_u := \{l\}$ 
12:   copy  $C_l$  into  $P_{u,l}^+$ 
13:    $P_{u,l}^- := \emptyset$ 
14: else
15:    $L_u := \emptyset$ 
```

non-neighbors of x in C_l . Note that all the variables are uniquely determined when G , X and the labels for all clusters are fixed.

For example, look at the graph depicted in Fig. 1. $X = \{x, y\}$ is a 3-approximate solution, and C_1 and C_2 are clusters. Here, the set of cluster labels is $\{1, 2\}$, $l_a = l_b = l_c = l_d = 1$ and $l_e = l_f = 2$. $L_x = \{1\}$ and $L_y = \{1, 2\}$. $P_{x,1}^+ = \{a, c\}$, $P_{x,1}^- = \{b, d\}$, $P_{y,1}^+ = \{c\}$, $P_{y,1}^- = \{a, b, d\}$, $P_{y,2}^+ = \{e, f\}$ and $P_{y,2}^- = \{\}$.

4.3 Algorithm

4.3.1 Update

Let us explain how to update the data structure when an edge is added or removed. Before describing the whole algorithm, let us explain subroutines used in the algorithm. Algorithm 2 is used to add a vertex u in $V \setminus X$ to X , and Algorithm 3 is to remove a vertex y from X under the condition that $X \setminus \{y\}$ is still a cluster vertex deletion. Given a cluster vertex deletion X , Algorithm 4 computes a 3-approximate solution X' .

Lemma 2. *Algorithm 2 adds a vertex u to X and updates the data structure correctly in $O(|X| \log n)$ time.*

Proof. At line 1, l is the label of the cluster that the vertex u belongs to. By removing u from C_l at line 2, C_l is correctly updated.

At line 3, we iterate over all $x \in X$ such that x is incident to the cluster C_l . For the other vertices in X , since all clusters except C_l are not changed, we need no updates. If there is an edge between x and u , u is in $P_{x,l}^+$. Thus we remove u from the set to correctly update $P_{x,l}^+$ (line 5). If $P_{x,l}^+$ becomes empty by the operation, it means that C_l is no longer incident to x , and thus we remove l from L_x (line 6). If there is no edge between x and u , u is in $P_{x,l}^-$. Thus we remove u from the set to correctly update $P_{x,l}^-$ (line 8). At line 9, we add u to X and complete the update of X .

Algorithm 3 remove $y \in X$ from X assuming $G[V \setminus (X \setminus \{y\})]$ is a cluster graph

```

1: remove  $y$  from  $X$ 
2: if  $L_y = \emptyset$  then
3:    $l_y :=$  new label
4:    $C_{l_y} := \{y\}$ 
5: else
6:    $|L_y|$  must be one. Let  $l_y$  be the unique element in  $L_y$ .
7:   add  $y$  to  $C_{l_y}$ 
8:  $l := l_y$ 
9: for all  $x \in X$  such that  $l \in L_x$  do
10:  if  $\{x, y\} \in E$  then add  $y$  to  $P_{x,l}^+$ 
11:  else add  $y$  to  $P_{x,l}^-$ 
12: for all  $x \in X$  such that  $y \in N(x)$  and  $l \notin L_x$  do
13:  add  $l$  to  $L_x$ 
14:   $P_{x,l}^+ := \{y\}$ 
15:  copy  $C_l$  into  $P_{x,l}^-$  and remove  $y$  from  $P_{x,l}^-$ 

```

Finally we initialize L , P^+ and P^- for u (line 10 to 15). If C_l is now empty, u is not incident to any cluster, and thus L_u is initialized as an empty set. Otherwise, u is incident only to the cluster C_l . Thus we initialize L_u as $\{l\}$ (line 11). Since $C_l \cup \{u\}$ was a cluster, u is incident to every vertex in C_l . Thus we initialize $P_{x,l}^+$ copying C_l , and initialize $P_{x,l}^-$ to be empty set (line 12 and 13). Using a persistent data structure, copying C_l into $P_{x,l}^+$ can be done in $O(1)$ time.

Removing a vertex from $P_{x,l}^+$ or $P_{x,l}^-$ (line 5 and 8) takes $O(\log n)$ time, and this part is repeated at most $|X|$ times. This is the dominant part of the algorithm. Thus the time complexity of the algorithm is $O(|X| \log n)$. \square

Lemma 3. *If $G[V \setminus (X \setminus \{y\})]$ is a cluster graph, Algorithm 3 removes a vertex y from X and updates the data structure correctly in $O(|X| \log n)$ time.*

Proof. First, we remove y from X and complete the update of X .

From line 2 to line 7, we compute l_y , the label of the cluster that the vertex y belongs to, and update C_{l_y} . Note that for any cluster label $l' \neq l_y$, $C_{l'}$ is not affected by the removal of y . If L_y is empty, it means that there is not adjacent vertex of y in $V \setminus X$, and thus we create a new cluster label for the cluster $\{y\}$. Otherwise, from the assumption that $G[V \setminus (X \setminus \{y\})]$ is a cluster graph, y is adjacent to exactly one cluster. Let l_y be the unique label in L_y , and then we add y to C_{l_y} .

Let l be l_y for notational brevity (line 8). We update the values L_x , $P_{x,l}^+$ and $P_{x,l}^-$ for each $x \in X$. Again, note that for any cluster label $l' \neq l$, $P_{x,l'}^+$ and $P_{x,l'}^-$ are not changed by the removal of y . If $l \in L_x$, or equivalently x is already adjacent to the cluster C_l before y is added to C_l , then we add y to $P_{x,l}^+$ or $P_{x,l}^-$ according to whether y is adjacent to x or not. If $l \notin L_x$, then x is not adjacent to any vertex in C_l before y is added to. If x is also not adjacent to y , no updates are needed. If x is adjacent to y , then x is incident to the cluster C_l after the removal of y from X . Thus we add l to L_x , and initialize $P_{x,l}^+$ as $\{y\}$ and $P_{x,l}^-$ as $C_l \setminus \{y\}$. To create $P_{x,l}^-$, we copy C_l into $P_{x,l}^-$ and remove y from $P_{x,l}^-$. By using a persistent data structure, the update can be done in $O(\log n)$ time.

Algorithm 4 compute a new 3-approximate solution X'

```

1:  $V' := \emptyset$ 
2:  $X_0 := \emptyset$ 
3: for all  $x \in X$  do
4:   if  $|L_x| > |X| + 1$  then
5:     add  $x$  to  $X_0$ 
6:   else
7:     add  $x$  to  $V'$ 
8:     for all  $l \in L_x$  do
9:       take  $\min(|P_{x,l}^+|, |X| + 1)$  vertices from  $P_{x,l}^+$ , and add them to  $V'$ 
10:      take  $\min(|P_{x,l}^-|, |X| + 1)$  vertices from  $P_{x,l}^-$ , and add them to  $V'$ 
11:  $Y :=$  3-approximate cluster vertex deletion of  $G[V']$ 
12: if  $|Y| > |X \setminus X_0|$  then  $X' := X$ 
13: else  $X' := X_0 \cup Y$ 

```

It is easy to check that the time complexity of the algorithm is $O(|X| \log n)$. □

Lemma 4. *Algorithm 4 computes a 3-approximate solution in $O(|X|^8)$ time.*

In order to prove Lemma 4, let us prove Lemma 5 and 6.

Lemma 5. *Let V' and X_0 be the sets computed by Algorithm 4. If $S \subseteq V'$ is a cluster vertex deletion of $G[V']$ such that $|S| \leq |X \setminus X_0|$, then $S \cup X_0$ is a cluster vertex deletion of G .*

Proof. Assume that S is not a cluster vertex deletion of $G[V \setminus X_0]$. This implies that there is an induced P_3 in $G[(V \setminus X_0) \setminus S]$. Let x, y be vertices in $X \setminus X_0$ and u, v be vertices in $V \setminus (X \cup S)$. There are four possible types of induced paths: (1) xuy , (2) xyu , (3) xuv , and (4) uxv . We will rule out all these cases by a case analysis (see Fig. 2).

- (1) Let $A = \{w \in V' \cap C_{l_u} \mid xw \in E \wedge yw \notin E\}$, $B = \{w \in V' \cap C_{l_u} \mid xw \notin E \wedge yw \in E\}$ and $C = \{w \in V' \cap C_{l_u} \mid xw \in E \wedge yw \in E\}$. By the construction of V' , $|A| + |C| \geq |X| + 1$ and $|B| + |C| \geq |X| + 1$. Thus $\min\{|A|, |B|\} \geq |X| - |C| + 1$. Since $x, y \notin S$ and $\{x, y\} \notin E$, S must contain $C \cup B$ or $C \cup A$. Thus $|S| \geq |X| + 1$, which is a contradiction.
- (2) Let $A = \{w \in V' \cap C_{l_u} \mid xw \notin E \wedge yw \notin E\}$, $B = \{w \in V' \cap C_{l_u} \mid xw \in E \wedge yw \in E\}$ and $C = \{w \in V' \cap C_{l_u} \mid xw \notin E \wedge yw \in E\}$. By the construction of V' , $|A| + |C| \geq |X| + 1$ and $|B| + |C| \geq |X| + 1$. Thus $\min\{|A|, |B|\} \geq |X| - |C| + 1$. Since $x, y \notin S$ and $\{x, y\} \in E$, S must contain $C \cup B$ or $C \cup A$. Thus $|S| \geq |X| + 1$, which is a contradiction.
- (3) Since $|S| \leq |X|$, there is a vertex $u' \in (V' \cap C_{l_u}) \setminus S$ such that $\{x, u'\} \in E$ and a vertex $v' \in (V' \cap C_{l_u}) \setminus S$ such that $\{x, v'\} \notin E$. However it contradicts the fact that $G[V' \setminus S]$ contains no induced P_3 .
- (4) Since $|S| \leq |X|$, there is a vertex $u' \in (V' \cap C_{l_u}) \setminus S$ such that $\{x, u'\} \in E$ and a vertex $v' \in (V' \cap C_{l_u}) \setminus S$ such that $\{x, v'\} \in E$. However it contradicts the fact that $G[V' \setminus S]$ contains no induced P_3 .

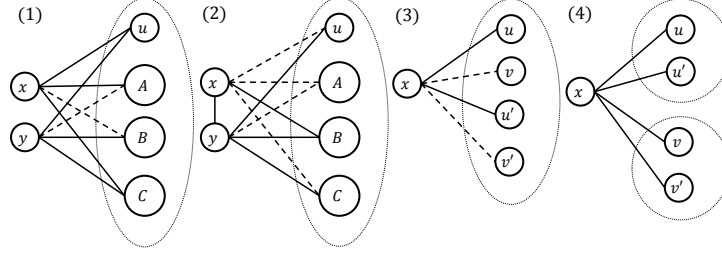


Figure 2: Case analysis in the proof of Lemma 5. A dotted line denotes there is no edge(s)

□

Lemma 6. *Let V' and X_0 be the sets computed by Algorithm 4. For any cluster vertex deletion T of G such that $|T| \leq |X|$, the following hold:*

1. T contains X_0 ,
2. $T \cap V'$ is a cluster vertex deletion of $G[V']$.

Proof. First, let us prove that T contains X_0 . Assume there exists $x \in X_0 \setminus T$. Since $|L_x|$, the number of adjacent clusters of x , is more than $|X| + 1$, in order to avoid induced P_3 , T must contain at least $|L_x| - 1 > |X|$ vertices from adjacent clusters. It contradicts the fact that $|T| \leq |X|$. Thus, T contains X_0 , and so $T \setminus X_0$ is a cluster vertex deletion of $G[V \setminus X_0]$.

Since $G[V \setminus T]$ is a cluster graph, its induced subgraph $G[V' \setminus T]$ is also a cluster graph. Thus $T \cap V'$ is a cluster vertex deletion of $G[V']$. □

Proof of Lemma 4. Let X^* be a minimum cluster vertex deletion. Since X is a cluster vertex deletion, we have $|X^*| \leq |X|$. By Lemma 6, it holds that $X_0 \subseteq X^*$, and $X^* \setminus X_0$ is a cluster vertex deletion of $G[V']$. $X^* \setminus X_0$ is actually a minimum cluster vertex deletion of $G[V']$, because otherwise there is a cluster vertex deletion S of $G[V']$ such that $|S| < |X^* \setminus X_0| \leq |X \setminus X_0|$, but then by Lemma 5, $S \cup X_0$ becomes a cluster vertex deletion of G of size less than $|X^*|$, which is a contradiction.

If the size of the set Y computed at line 11 is larger than $|X \setminus X_0|$, the set X remains a 3-approximate solution. Otherwise, from Lemma 5, $Y \cup X_0$ is a cluster vertex deletion of G . Since Y is a 3-approximate solution and $X^* \setminus X_0$ is a minimum cluster vertex deletion of $G[V']$, we have

$$|Y \cup X_0| \leq 3|X^* \setminus X_0| + |X_0| \leq 3|X^*|. \quad (1)$$

Thus, $X' = Y \cup X_0$ is a 3-approximate solution on G .

The claimed time complexity is obtained as follows. The size of V' at line 11 is at most $2|X|(|X| + 1)^2 = O(|X|^3)$. The number of edges in the graph $G[V']$ is maximized when $G[V' \setminus X]$ is composed of $|X| + 1$ cliques with size $|X|(|X| + 1)$. Thus the number of edges is at most $|X|^2(|X| + 1)^3 = O(|X|^5)$. Thus, a 3-approximate solution can be computed in $O(|X|^8)$ time using the trivial algorithm described in Section 4.1, and thus the claimed time complexity holds. □

Now we are ready to describe how to update the data structure when an edge is modified. To add (remove) an edge $\{u, v\}$ to (from) a graph G , before modifying G , we add u and v to X one by one using Algorithm 2 unless the vertex is already in X . After the operation, we add (remove) the edge $\{u, v\} \subseteq X$ to (from) G . Note that we do not have to change our data structure by this operation. Now X is a cluster vertex deletion but may no longer be a 3-approximate solution. Then we compute a new 3-approximate solution X' using Algorithm 4.

Finally we replace X by X' as follows. Let R be $X \setminus X'$ and R' be $X' \setminus X$. We begin with adding every vertex in R' to X one by one using Algorithm 2. Then we remove every vertex in R from X one by one using Algorithm 3, and finish the replacement. During the process, X is always a cluster vertex deletion of the graph, and thus the assumption of Algorithm 3 is satisfied.

Let k be the maximum of the cvd numbers before and after the edge modification. During the above process, the size of X is increased to at most $6k$. Algorithm 4 is called only once, and Algorithm 2 and 3 are called $O(k)$ times. Thus together with Lemma 2, 3 and 4, the update time is $O(k^8 + k^2 \log n)$.

4.3.2 Query

Let us explain how to answer a query. To compute a minimum cluster vertex deletion X' , we use almost the same algorithm as Algorithm 4, but compute an exact solution Y at line 11 instead of an approximate solution. The validity of the algorithm can be proved by almost the same argument. The bottleneck of the algorithm is to compute a minimum cluster vertex deletion of the graph $G[V']$. Since the number of edges in $G[V']$ is $O(k^5)$ as noted in the proof of Lemma 4, using an $O(f(|G|, k))$ -time static algorithm for **Cluster Vertex Deletion**, we can obtain the solution in $O(f(k^5, k))$ time. For example, using the algorithm in [4], we can compute the solution in $O(1.9102^k k^5)$ time.

5 Another Dynamic Graph for Cluster Vertex Deletion (constant time update and query)

In this section, we give an $O(f(k))$ -time dynamic graph for **Cluster Vertex Deletion**, where k is the cvd number and f is some computable function. This algorithm is not efficient than the algorithm in Section 4 in many cases in practice. However, we introduce this algorithm here because it is asymptotically faster than the algorithm in Section 4 if k is a constant, and is a base of the dynamic graph for **Chromatic Number** parameterized by cvd number in Section 6.

5.1 Data Structure

We maintain X to be a minimum cluster vertex deletion. Initially $X = \emptyset$. Let us define an equivalence relation on $V \setminus X$ so that two vertices $u, v \in V \setminus X$ are equivalent if and only if u and v are in the same cluster and $N(u) \cap X = N(v) \cap X$. The important point is that we do not have to distinguish the vertices in the same class and can treat them as if they are one vertex weighted by the number of the vertices in the same class. To treat the vertices in a same class efficiently, we introduce an auxiliary data structure, an undirected graph H . The graph H is uniquely determined by G and X as follows (see Fig. 3.)

First, we introduce *cluster labels* L by assigning a different label $l \in L$ to a different cluster $C_l \subseteq V \setminus X$. For each cluster label $l \in L$ and a vertex set $S \subseteq X$, let $C_{l,S}$ denote the equivalent

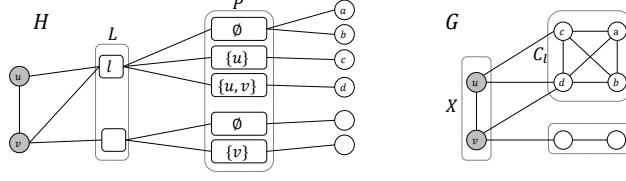


Figure 3: Correspondence of G and X to H .

class $\{v \in C_l \mid N(v) \cap X = S\}$. We introduce a *class label* $p_{l,S}$ for each nonempty equivalent class $C_{l,S}$. Let P be the set of the introduced class labels. The vertex set of H is $X \cup L \cup P \cup (V \setminus X)$.

Then, we add an edge between x and y in H if and only if one of the following conditions is satisfied:

- $x, y \in X$ and $e \in E(G[X])$,
- $x \in X, y \in L$ and $x \in N(C_y)$,
- $x \in L$ and $y = p_{x,S} \in P$ for some $S \subseteq X$, or
- $x = p_{l,S} \in P$ and $y \in C_{l,S}$.

5.2 Strategy

Whenever the graph G is updated, we transform the current solution X to a minimum solution as follows:

- (1) X is extended to be a solution.
- (2) A problem kernel $G[V']$ is obtained using the solution.
- (3) A minimum solution X' is obtained applying a (static) algorithm for $G[V']$, and X is exchanged for the minimum solution X' .

Let us call the steps (1), (2) and (3) as *growing phase*, *compression phase* and *exchange phase* respectively. To compute an exact solution, we can apply any exact FPT algorithm to the current problem kernel $G[V']$.

5.2.1 Move a vertex to X

Let us show how to move a vertex v in $V \setminus X$ to X and update the graph H efficiently. This procedure is used in the growing phase and the exchange phase.

- (1) Let $C_{l,S}$ be the equivalent class containing v . Remove the edge $\{p_{l,S}, v\}$ of H . If $C_{l,S}$ becomes empty, remove the label $p_{l,S}$ destroying the incident edge to l . Moreover, if C_l becomes empty, remove the label l destroying all incident edges to X .
- (2) If the label l is not removed, add the edge $\{v, l\}$. For each neighbor $u \in N_G(v) \cap X$, add the edge $\{u, v\}$. If $C_{l,S}$ becomes empty but C_l is still not empty, for each $u \in N_G(v) \cap X$ that is no more adjacent to the cluster C_l , remove the edge between u and l . Then, for each class label $p_{l,S'}$ adjacent to l , change its name to $p_{l,S' \cup \{v\}}$.

Step (1) corresponds to the removal of v from $V \setminus X$ and step (2) corresponds to the addition of v to X . It is easy to see that this procedure correctly updates H . Since there are at most $O(2^{|X|})$ equivalent classes for each cluster, the running time is $O(2^{|X|}|X|^2)$.

5.2.2 Growing Phase

To handle an insertion or a deletion of an edge, we first move its endpoints not in X to X to make X a cluster vertex deletion. This phase is completed in $O(2^k k^2)$ time, and now $|X|$ is at most $k + 2$.

5.2.3 Compression Phase

If we remove a vertex $v \in X$ from X that is adjacent to d different clusters, we have to include at least $d - 1$ vertices into X because there are no edges between different clusters. Thus, to improve the solution X , we can only remove a vertex $v \in X$ that is adjacent to at most $|X|$ different clusters. Let X_0 be the vertices in X such that the number of its adjacent clusters is at most $|X|$. As noted above, $X_1 = X \setminus X_0$ must be kept in X to improve the solution. Let L' be the labels of clusters adjacent to any vertex in X_0 . $|L'| \leq |X|^2$ by the definition of X_0 .

Now we make a vertex weighted graph G' . Let P_l denote the set of the class labels of the form $p_{l,*}$. The vertex set of G' is $X_0 \cup P'$, where $P' = \bigcup_{l \in L'} P_l$. Since the size of P_l is at most $2^{|X|}$, it holds $|V(H)| \leq |X| + 2^{|X|}|X|^2$. The weight $w : V(H) \rightarrow \mathbb{N}$ is defined by $w(v) = 1$ for every $v \in X_0$ and $w(p_{l,S}) = |C_{l,S}|$ for every $p_{l,S} \in P'$. For each edge $e \in E(G[X_0])$, we add e to G' . For each $v \in X_0$ and $p_{l,S} \in P'$ we add the edge between them if $v \in S$. Finally, for each $l \in L'$, we add edges between every two class labels in P_l , making $G'[P_l]$ a clique. This completes the construction of the graph G' .

Then we solve **Cluster Vertex Deletion** for the vertex weighted graph G' , and determine X' as the vertices corresponding to the solution. Since $|V(H)| \leq 2^k k^2 + k$, apparently it is solvable in $O(f(k))$ time for some function f . For example, using the $O(2^k k^9 + |V||E|)$ algorithm proposed by Hüffner, Komusiewicz, Moser and Niedermeier [17], we can solve the problem in $O(2^k k^9 + 8^k k^6) = O(8^k k^6)$ time.

5.2.4 Exchange Phase

Let R be $X \cap X'$. To exchange X for X' , we add all vertices in $X' \setminus R$ to X and then remove all vertices in $X \setminus R$ from X . Note that during the process, X is always a cluster vertex deletion. The additions to X are executed using the method in Section 5.2.1 at most k times.

Let us show how to remove a vertex v from X . Before removing v from X , we update H as follows.

- (1) Since $X \setminus \{v\}$ is a cluster vertex deletion, v is adjacent to at most one cluster label. If there is such a cluster label l , remove the edge $\{v, l\}$. Otherwise, introduce a new cluster label l corresponding to an empty cluster. Let S be $N_H(v)$, that is a subset of $X \setminus \{v\}$. Remove all edges from v to S .
- (2) Add the edges from the vertex in S to l unless it already exists. Then, add the edges $\{l, p_{l,S}\}$ and $\{p_{l,S}, v\}$, introducing the class label $p_{l,S}$ unless it already exists. Finally, we iterate over each class label $p_{l,S'}$ and change its name to $p_{l,S' \setminus \{v\}}$. Note that v is in S' by the assumption.

Step (1) corresponds to the removal of v from X and step (2) corresponds to the addition of v to $X \setminus V$. It is easy to see that this procedure takes $O(2^{|X|}|X|^2)$ time and correctly updates H .

During the exchange, the size of X is increased to at most $2k$. Thus the exchange phase can be done in $\sum_{k'=k+1}^{2k} O(k'^2 2^{k'}) = O(k^2 2^{2k})$ time.

In summary, we have obtained an $O(8^k k^6)$ -time dynamic graph algorithm for Cluster Vertex Deletion. The dominant part is the compression phase.

6 Dynamic Graph for Chromatic Number Parameterized by CVD Number

Let $G = (V, E)$ be a graph. Given a vertex set $S \subseteq V$, $\Pi(S)$ denotes the set of all *partitions* of S . The size of all partitions $|\Pi(S)|$ is called *Bell number* $B_{|S|}$ and it is known that $B_n = O((\frac{0.792n}{\ln(n+1)})^n)$ [1]. Note that we can easily enumerate all partitions in $\Pi(S)$ in $O(B_{|S|})$ time. A partition of the vertices of a graph can be regarded as a *coloring* of the graph, since given a partition, by coloring vertices in the same block with the same color, we can determine the corresponding coloring on the graph. For $p \in \Pi(S)$ and $S' \subseteq S$, let $p|_{S'}$ denote the restriction of p into S' , that is the unique coloring $p' \in \Pi(S')$ that can be extended to p . For a vertex set $S \subseteq V$, we call a partition $p \in \Pi(S)$ a *proper coloring* or *proper* if no two vertices in S sharing the same edge are in a same block. In other words, considering the corresponding coloring, if there are no adjacent vertices with the same color, then the partition is a proper coloring. For a coloring p , its *size* $|p|$ is defined by the number of blocks of p . **Chromatic Number** is a problem of finding the size of a minimum proper coloring of V . The size is called *chromatic number*.

In this section, we provide a data structure for **Chromatic Number**, that works efficiently when the size of a minimum cluster vertex deletion (cvd number) of the graph is small.

Theorem 3. *There is a data structure treating the graph G that supports the following operations:*

- (1) *Compute the solution of Chromatic Number of the current graph G .*
- (2) *Add an edge to G .*
- (3) *Remove an edge from G .*

The operation (1) takes $O(1)$ time, and both (2) and (3) take $O(\log n)$ time assuming the cvd number is a constant. More precisely, the time complexity of (1) is $O(1)$, and of (2) and (3) are $O(B_{2k}(4^k k^3 + \log n))$, where k is maximum of the cvd number of the current graph and cvd number of the updated graph.

The first static FPT algorithm for this problem is proposed by Martin Doucha and Jan Kratochvíl [7]. Our algorithm is based on the algorithm, but updates necessary information dynamically.

6.1 Data Structure

As an underlying data structure, we maintain the data structure described in Section 5. For a cluster label l , let X_l be $\{x \in X \mid N(x) \cap C_l \neq \emptyset\}$. X_l is easily computed in $O(k)$ time from the auxiliary data structure H . In addition, we maintain the following information:

- For every cluster label l and coloring $p \in \Pi(X_l)$, $\chi_{l,p}$ is the size of a minimum proper coloring on $G[X_l \cup C_l] - E(G[X_l])$ made by extending p .
- For every $Y \subseteq X$ and $p \in \Pi(Y)$, Λ_p is the multiset containing every $\chi_{l,p}$ such that $X_l = Y$. We use a balanced binary search tree to hold Λ_p to efficiently remove or add a value and get the maximum value in the set.

More formally, $\chi_{l,p}$ is defined by

$$\min\{|q| \mid \begin{array}{l} q \in \Pi(C_l \cup X_l) \wedge q|_{X_l} = p \\ \wedge q \text{ is a proper coloring on } G[X_l \cup C_l] - E(G[X_l]) \end{array}\}, \quad (2)$$

and for $Y \subseteq X$ and $p \in \Pi(Y)$, Λ_p is the multiset

$$\{\chi_{l,p} \mid l \text{ is a cluster label} \wedge X_l = Y\}. \quad (3)$$

When all Λ_p are given, we can compute the chromatic number of the graph as:

$$\min_{\text{proper coloring } p \in \Pi(X) \text{ on } G[X]} \max\{|p|, (\max_{Y \subseteq X} \text{maximum value in } \Lambda_{p|_Y})\}, \quad (4)$$

where maximum value of an empty set is defined to be 0.

Proof.

$$\max\{|p|, (\max_{Y \subseteq X} (\max \Lambda_{p|_Y}))\} \quad (5)$$

is the minimum number of colors needed to color the graph $G - E(G(X))$, where the coloring on X is fixed to be p . Thus moving p over all proper coloring on $G[X]$ and taking the minimum of (5), we obtain the size of a minimum proper coloring. \square

Since there are at most $n = |V|$ clusters, the maximum value in $\Lambda_{p|_Y}$ is obtained in $O(\log n)$ time. There are at most $O(B_{|X|}|X|)$ possibilities of $p|_Y$ in the formula (4). Thus the formula is computed in $O(B_{|X|}|X| \log n)$ time.

6.2 Algorithm to Update Data Structure

Let us explain how to update the data structure when an edge is added or removed. Before describing the whole algorithm, let us introduce subroutines used in the algorithm.

6.2.1 Add (Remove) a Vertex to (from) X .

Let us explain how to add (remove) a vertex to (from) X and update the data structure accordingly. We assume that when a vertex y is removed from X , $G[V \setminus (X \setminus \{y\})]$ is a cluster graph.

To add a vertex u to X :

1. Let l be the label of the cluster that the vertex u belongs to. For every $p \in \Pi(X_l)$, we remove $\chi_{l,p}$ from Λ_p to prepare the update of the value $\chi_{l,p}$.
2. We run the algorithm in Section 5. In particular, X , X_l , C_l and $C_{l,S}$ are updated for every $S \subseteq X$.

3. If the cluster C_l still exists, we compute $\chi_{l,p}$ for every $p \in \Pi(X_l)$ using Lemma 7, and then add $\chi_{l,p}$ to Λ_p .

To remove a vertex y from X :

1. From the assumption, y is adjacent to at most one cluster. If there is a cluster that y is adjacent to, let l be the label of the cluster, and we remove $\chi_{l,p}$ from Λ_p to prepare the update of the value $\chi_{l,p}$.
2. We run the algorithm in Section 5. In particular, X , X_l , C_l and $C_{l,S}$ are updated for every $S \subseteq X$, where l is the label of the cluster that y belongs to now.
3. We compute $\chi_{l,p}$ for every $p \in \Pi(X_l)$ using Lemma 7, and then add $\chi_{l,p}$ to Λ_p .

Now, we prove that $\chi_{l,p}$ can be computed efficiently.

Lemma 7. *After the equivalent class $C_{l,S}$ is updated for each $S \subseteq X$, for any $p \in \Pi(X_l)$, $\chi_{l,p}$ can be computed in $O(2^{|X|}|X|^3)$ time.*

Proof. We want to find the size of a minimum proper coloring q on $G[X_l \cup C_l] - E(G[X_l])$ such that $q|_{X_l} = p$. Without loss of generality, let the set of colors used for X_l be $\{1, \dots, |p|\}$. For $S \subseteq X_l$, let $c(S) \subseteq \{1, \dots, |p|\}$ denote the colors assigned to S .

Since $G[C_l]$ is a cluster, the colors assigned to C_l must be distinct. Minimizing the size of the coloring is equivalent to maximizing the number of vertices in C_l that are colored with $1, \dots, |p|$.

Let r be the number of vertices in C_l that are assigned a color in $\{1, \dots, |p|\}$. We want to maximize r to minimize the number of colors $|p| + |C_l| - r$.

We compute the maximum possible r by constructing a graph $(\{s\} \cup L \cup R \cup \{t\}, F)$ and computing the size of a maximum flow from s to t . Create vertices $x_1, \dots, x_{|p|}$ and let L be $\{x_1, \dots, x_{|p|}\}$. We add edges $\{s, x_1\}, \dots, \{s, x_{|p|}\}$ with capacity one. For each $S \subseteq X$, we create a vertex y_S and add y_S to R , and add the edge $\{y_S, t\}$ with capacity $|C_{l,S}|$. Then, for each $S \subseteq X$ and $i \in \{1, \dots, |p|\} \setminus c(S)$, we add an edge between x_i and y_S with capacity one, completing the construction of the graph. We compute r as the size of a maximum s - t flow of the constructed graph, and conclude $\chi_{l,p}$, the size of a minimum coloring on $G[X_l \cup C_l] - E(X_l)$ obtained by extending p , is $|p| + |C_l| - r$.

The size of L is at most $|X|$ and the size of R is at most $2^{|X|}|X|$. Thus the number of edges in the graph is at most $O(2^{|X|}|X|^2)$. Since the size of a maximum flow is at most $|L| \leq |X|$, using Ford-Fulkerson algorithm [12]², we compute the solution in $O(|F||X|) = O(2^{|X|}|X|^3)$ time. \square

Since u was not adjacent to any cluster other than C_l , no change of $\chi_{l',p'}$ is needed for any cluster label $l' \neq l$ and $p' \in \Pi(X_{l'})$. Thus we have obtained the following lemma.

Lemma 8. *We can add (remove) a vertex to (from) X and update the data structure accordingly in $O(B_{|X|} \log n + B_{|X|} 2^{|X|}|X|^3)$ time.*

$O(B_{|X|} \log n)$ is the time to remove (add) $\chi_{l,p}$ from (to) Λ_p for every $p \in \Pi(X_l)$, and $O(B_{|X|} 2^{|X|}|X|^3)$ is the time to compute $\chi_{l,p}$ for every $p \in \Pi(X_l)$.

²whose time complexity is $O((\text{the number of edges})(\text{maximum flow size}))$

6.2.2 Add or Remove an Edge

Now we are ready to describe how to update the data structure when the graph is modified.

As we did in Section 5, before the edge $\{u, v\}$ is added to or removed from G , we move u and v to X making X a cluster vertex deletion of the modified graph. When the edge is actually added to (removed from) G , no change of the data structure is needed since any edge lying on X does not affect $\chi_{l,p}$ for any cluster label l and $p \in \Pi(X_l)$.

After the edge is added to (removed from) G , we compute a minimum cluster vertex deletion X' using the algorithm in Section 5. Then we replace X with X' . Let R be $X \setminus X'$ and R' be $X' \setminus X$. We begin with adding every vertex in R' to X one by one. Then we remove every vertex in R from X one by one finishing the replacement. During the process, X is always a cluster vertex deletion of the graph, and thus the algorithm works correctly. As the size of X is increased to at most $2k$, by Lemma 8, the replacement is completed in $\sum_{k'=k}^{2k} O(B_{k'} \log n + B_{k'} 2^{k'} k'^3) = O(B_{2k}(\log n + 4^k k^3))$ time.

After all information is updated, finally we compute the solution of **Chromatic Number** using the formula (4). As noted above, it is computed in $O(B_{|X|}|X| \log n) = O(B_k k \log n) = O(B_{2k} \log n)$ time.

Putting things altogether, now we have proved Theorem 3.

7 Dynamic Graph for Bounded-Degree Feedback Vertex Set

A vertex set is called *feedback vertex set* if its removal makes the graph a forest. **Feedback Vertex Set** is the problem of finding a minimum feedback vertex set. In this section, we assume every vertex in G always has degree at most d . We maintain X to be a minimum feedback vertex set. Initially $X = \emptyset$. As noted in the Introduction, our algorithm is based on the static algorithm by Guo, Gramm, Hüffner, Niedermeier and Wernicke [14].

7.1 Data Structure

The key point is that we keep the forest $G[V \setminus X]$ using dynamic tree data structures called *link-cut tree* data structures presented by Sleator and Tarjan [29]. Link-cut tree is a classic data structure that supports many operations on a forest in $O(\log n)$ amortized time. We exploit the following operations of link-cut trees. All of them are amortized $O(\log n)$ -time operations:

- **link**(r, v): If the vertex r is a root of a tree and the vertices r and v are in different trees, add an edge from r to v .
- **cut**(u, v): Remove the edge between u and v .
- **evert**(v): Make the vertex v a root of the tree containing v .
- **root**(v): Return the root of the tree containing v .
- **nca**(u, v): If the vertices u and v are in the same tree, find the nearest common ancestor of u and v in the rooted tree containing u and v .
- **parent**(v): If v is not a root, return the parent of v .

For further understanding, see [29]. Note that we can check whether the vertices v and u are in the same tree testing if **root**(v) equals **root**(u) or not.

7.2 Strategy

As with Section 5.2, whenever the graph G is updated, we transform the current solution X to a minimum solution as follows:

- (1) X is extended to be a solution.
- (2) A problem kernel $G[V']$ is obtained using the solution.
- (3) A minimum solution X' is obtained applying a (static) algorithm for $G[V']$, and X is exchanged for the minimum solution X' .

Let us call the steps (1), (2) and (3) *growing phase*, *compression phase* and *exchange phase* respectively. To compute an exact solution, we can apply any exact FPT algorithm for current problem kernel $G[V']$.

7.3 Growing Phase

When an edge $e = \{u, v\}$ is added to G , we insert one of its endpoints, say u , to current solution. That is, we add u to X and remove u from the link-cut tree containing u calling **cut**(u, v) for each v in $N(u) \setminus X$. The time complexity of this phase is $O(d \log n)$. When an edge is removed from G , we do nothing. After the operation, X is a feedback vertex set and the size of X is at most $k + 1$.

7.4 Compression Phase - Outer Loop

In our algorithm, we only consider special kinds of solutions of Feedback Vertex Set. We call them *maximum overlap solutions*.

Definition 1 (Maximum Overlap Solution). *For given graph G and a feedback vertex set X of G , $X' \subseteq V$ is called maximum overlap solution if X' is a minimum feedback vertex set of G and the size of intersection $|X \cap X'|$ is the maximum possible.*

At least one maximum overlap solution apparently always exists. Then, we try to find a maximum overlap solution X' . To do this, we use the idea of exhaustively considering all possible intersections of $X \cap X'$. That is, we compute a minimum feedback vertex set X'' that satisfies $X \cap X'' = R$ for all $2^{|X|}$ possibilities of $R \subseteq X$, and let X' be the minimum of them. Now our job is to solve the following task.

Task 1 (Disjoint Feedback Vertex Set). *Given a feedback vertex set X and a subset R of X , find a minimum vertex set S' such that S' is a feedback vertex set of $G[V \setminus R]$ and $S \cap S' = \emptyset$, where $S = X \setminus R$, or output 'NO' if there is no such S' .*

The important point is that for the purpose of solving Feedback Vertex Set, it is sufficient to obtain the algorithm such that (1) if the algorithm outputs a vertex set S' , it is a correct answer and (2) if there is a maximum overlap solution X' such that $X \cap X' = R$, the algorithm outputs a correct answer, i.e., the algorithm can incorrectly output 'NO' if there is no maximum overlap solution X' such that $X \cap X' = R$. Thus we consider such an algorithm.

7.5 Compression Phase - Inner Loop in $O(f(k)|E|)$ time

First, let us introduce an $O(f(k)|E|)$ algorithm. Let V' be $V \setminus R$. We reduce the graph $G[V']$ applying the following two rules recursively.

1. If there is a degree 1 vertex $v \in V' \setminus S$, remove v from the graph
2. If there is a degree 2 vertex $v \in V' \setminus S$, remove v and connect its neighbors by an edge. It may make parallel edges between the neighbors.

Note that after the reduction every vertex not in S has degree at least three. Let **reduced**(R) denote the (uniquely determined) reduced graph (see Fig. 4).

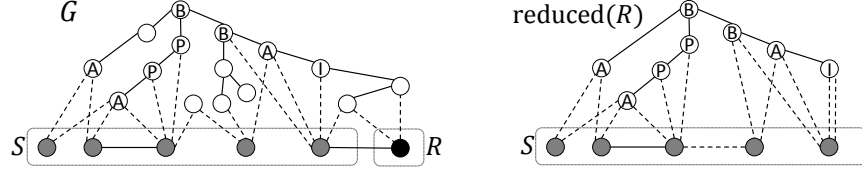


Figure 4: Example of the original graph G and the reduced graph. The white vertices with some letters are core vertices, and each letter denotes the class of the vertex defined in the proof of Lemma 10.

After the reduction, we try to find a minimum feedback vertex set disjoint from S in the reduced graph.

This reduction is based on the iterative compression algorithm in [14]. The differences are: (1) we do not reduce the graph even if there are parallel edges, and (2) we contract the vertex v even if $N(v) \subseteq S$. The second difference may lead to the algorithm that incorrectly outputs ‘NO’. However, by the following lemma, such cases never happen when R is a correct assumption.

Lemma 9. *In the setting of Task 1, if there is a maximum overlap solution X' such that $R = X \cap X'$, **reduced**(R) has a feedback vertex set S' such that $S' \cap S = \emptyset$ and $|S'| + |R| = |X'|$.*

Proof. We prove that any vertex in $X' \setminus R$ never removed during the reduction, and the lemma follows since $X' \setminus R$ becomes a minimum feedback vertex set of **reduced**(R). Let us hypothesize that during the reduction, a vertex v in $X' \setminus R$ has been removed. It means that there are at most two edge disjoint paths from v to vertices in S . If there is no path from v to S , $X' \setminus \{v\}$ must be a feedback vertex set of G and contradict the minimality of X' . Otherwise let $u \in S$ be the end point of a path from v to S . Then $X' \setminus \{v\} \cup \{u\}$ must be also a minimum feedback vertex set and contradict the maximality of $|X \cap X'|$. \square

Thus, to solve **Feedback Vertex Set** for the original graph, we only have to solve **Feedback Vertex Set** for the reduced graph. In fact, if the reduced graph has a smaller solution, the graph must be not so large. Let us call the vertex set $V(G') \setminus S$ *core vertices*. As in [14], we can prove the following lemma.

Lemma 10. *In the setting of Task 1, if there is a maximum overlap solution X' such that $R = X \cap X'$, the size of the core vertices is no more than $13|S|$.*

Proof. Let G' be **reduced**(R) and V' be core vertices $V(G') \setminus S$. By Lemma 9 it holds $(X' \setminus R) \subseteq V'$. We classify the vertices V' as follows (see Fig. 4):

- $A = \{v \in V' \mid |\delta_{G'}(\{v\}, S)| \geq 2\}$.
- $B = \{v \in V' \setminus A \mid |N(v) \cap (V' \setminus S)| \geq 3\}$.
- $C = V' \setminus (A \cup B)$.

We prove upper bounds for A , B and C separately.

To prove the upper bound for A , we consider the subgraph $G_A = (A \cup S, \delta_{G'}(A, S))$ for G' . If $A \cup S$ is a forest, $|E(G_A)| < |V(G_A)|$. Thus $2|A| \leq |E(G_A)| < |V(G_A)| = |S| + |A|$ and therefore $|A| < |S|$. Since we can make G' a forest removing at most $|S|$ vertices disjoint from S , $|A| < 2|S|$.

To prove the upper bound for B , we consider the forest $G[V']$. Observe that all leaves of the forest are from A . In fact if v is a leaf of the forest, $d_{G[V']}(v) \leq 1$, and since $d_{G'}(v) \geq 3$, $|\delta_{G'}(\{v\}, S)| \geq 2$ must hold. Each vertex in B is an internal node of degree at least three in $G[V']$. The number of such vertices cannot be more than the number of the leaves, therefore $|B| \leq |A| < 2|S|$.

Each vertex v in C has a degree two in $G'[V']$ and exactly one incident edge to S . Hence, $G[C]$ is composed of paths and isolated vertices. Let P be the path vertices and I be the isolated vertices. We separately bound the number of P and I .

For each isolated vertex $v \in I$, the number of edges between v and $A \cup B$ is exactly two. Since $G[V']$ is forest, $2|I| \leq |E(G[A \cup B \cup I])| < |A \cup B \cup I| < 4|S| + |I|$, and therefore $|I| < 4|S|$.

To prove the upper bound for P , we consider the graph $G[S \cup P]$. There are exactly $|P|$ edges between S and P and at least $|P|/2$ edges among $G[P]$. Thus the number of edges in $G[S \cup P]$ is at least $3|P|/2$. If it is a forest, $3|P|/2 \leq |E(G[S \cup P])| < |V(G[S \cup P])| = |S| + |P|$, hence $|P| < 2|S|$. Since we can make G' a forest removing at most $|S|$ vertices disjoint from S and removing a vertex evicts at most three vertices from P , we obtain that $|P| < 5|S|$.

Altogether, $|V'| = |A| + |B| + |I| + |P| < |S| + 2|S| + 2|S| + 4|S| + 5|S| = 13|S|$. \square

Using Lemma 9 and 10, we can solve Task 1. First we construct the reduced graph in $O(|E|)$ time. Then if the size of the graph is more than $14|S|$, we safely return ‘NO’, and otherwise we solve Disjoint Feedback Vertex Set for the reduced graph in $O(f(k))$ time using some algorithm. When k is small or a fixed constant, the bottleneck is the part of reducing the graph in $O(|E|)$ time. To speed up the part, we make use of operations on link-cut trees.

7.6 Faster Reduction

Let G' denote **reduced**(R). In this section, we show the algorithm computing G' in $O(k^3 d^3 \log n)$ amortized time by finding the core without explicitly reducing the graph. The graph $G[V \setminus X]$ is a forest. For vertices $u, v, w \in V \setminus X$ that are in the same tree, let **meet**(u, v, w) denote the vertex at which the path from v to u and the path from w to u firstly meet. In other words, **meet**(u, v, w) is **nca**(v, w) on the tree rooted at u , thus is computable in $O(\log n)$ amortized time.

To generate the core vertices C , we iterate over every set of three edges $\{e_1, e_2, e_3\} \subseteq \delta(S, V \setminus X)$ and add **meet**(v_1, v_2, v_3) to C , where v_i is the endpoint of e_i that is in $V \setminus X$.

Lemma 11. *The vertices C generated by the above algorithm is equal to core vertices, and the algorithm runs in $O(k^3 d^3 \log n)$ amortized time.*

Proof. The time complexity is straightforward from the above discussion. Let v be a vertex in $V \setminus X$. The vertex v is a core vertex if and only if there are at least three edge disjoint paths from v to S that contains no internal vertex in X . If v is a core vertex and there are such paths P_1, P_2 and P_3 to $s_1, s_2, s_3 \in S$, v must be in C since v is the vertex at which the path from s_1 to s_3 through P_1 and P_3 and the path from s_2 to s_3 through P_2 and P_3 firstly meet. Conversely, if v is in C , it directly means there are at least three edge disjoint paths from v to S that contains no internal vertex in X . \square

After finding the core vertices, we can also compute the edges in G' in $O(k^3 d^3 \log n)$ amortized time using link-cut tree operations as follows.

Let us construct the graph G'' to be the reduced graph $G' = \text{reduced}(R)$. The vertex set of G'' is $C \cup S$.

Firstly, let us consider the edges in $G'[C]$. We iterate over every $u, v \in C$. The edge $\{u, v\} \subseteq C$ is in G' if and only if there is a path from u to v in the forest $G[V \setminus X]$ and it contains no internal vertex in C . There is a path from u to v if and only if u and v are in the same tree. To check if the path contains an internal vertex in C , firstly we make the vertex u a root of the tree containing u calling **event**(u). Then we iterate over every $w \in C$ without u and v . For each $w \in C$, we cut the edge between w and its parent and test if u and v are still in the same tree. If u and v are now separated, it means w were on the path from u to v . After checking, we restore the cut edge. If there is no internal vertex w , we add the edge $\{u, v\}$ to G'' . Since $|C| \leq 14|S|$, using the algorithm, we complete the construction of $G''[C]$ in $O(k^3 \log n)$ amortized time.

Let us consider the edges between S and C . Observe that each edge from $s \in S$ to $v \in C$ in G' corresponds to a path from s to v that have no internal vertex in $C \cup S$. Using the algorithm similar to the previous one, we can compute the edges in $O(k^3 d \log n)$ time. Firstly we iterate over every $\{s, u\} \in \delta_G(S, V \setminus X)$ and $v \in C$, where $s \in S$ and $u \in V \setminus X$. We call **event**(u), and for each $w \in C$ without u , we check if w is an internal vertex of the path from s to u in $O(\log n)$ amortized time. If there is no internal vertex in C , we add the edge $\{s, v\}$ to G'' . Since $|\delta_G(S, V \setminus X)| = O(kd)$, using the algorithm, we complete the construction of the edges between S and C in $O(k^3 d \log n)$ amortized time.

Finally let us consider the edges in S . We add every edge in $G[S]$ to G'' in $O(k^2)$ time. Furthermore, for each $s, t \in S$, if there is a path of length more than 1 from s to t whose internal vertices not containing a vertex in $C \cup S$, it becomes an edge between s and t . To find the paths, we iterate over every set of two edges $\{e_1, e_2\} \subseteq \delta(S, V \setminus X)$. Let $e_i = \{s_i, v_i\}$, $s_i \in S$ and $v_i \in V \setminus X$. If v_1 and v_2 are in the same tree, there is a path P from v_1 to v_2 in $G[V \setminus X]$. To check if P contains core vertex or not, we iterate over every edge e_3 in $\delta(S, V \setminus X)$ without e_1 and e_2 . Let v_3 be the endpoint of e_3 that is in $V \setminus X$. If v_3 and v_1 is in the same tree, we can conclude P contains core vertex **meet**(v_1, v_2, v_3). If there is no such edge, we can conclude P contains no core vertex by the construction of the core vertices, and add $\{v_1, v_2\}$ to G'' . It runs in $O(k^3 d^3 \log n)$ amortized time.

Now, we have completed the construction of the reduced graph $G'' = G'$. Altogether, we can construct the reduced graph in $O(k^3 d^3 \log n)$ amortized time.

Now we are ready to solve Task 1. First, we compute $G' = \text{reduced}(R)$ in $O(k^3 d^3 \log n)$ amortized time. If $|V(G')| \geq 14|S|$, we return 'NO'. Otherwise, for example, using the $O(3.83^k k |V|^2)$ -time algorithm proposed by Cao, Chen and Liu [3], we solve the problem in $O(3.83^k k^3)$ time.

Together with the outer loop cost $O(2^k)$, we finish the compression phase in $O(7.66^k k^3 + 2^k k^3 d^3 \log n)$ amortized time.

7.7 Exchange Phase

Given a minimum feedback vertex set X' , the remaining work is to exchange the solution X to X' . Let R be $X \cap X'$, S be $X \setminus R$ and S' be $X' \setminus R$. First, we add every vertex in S' to X as in Section 7.3. Then, for each vertex u in S we remove u from X and add u to the link-cut trees calling **evert**(v) and **link**(u, v) for each v in $N(u) \setminus X$. Since $|X| \leq k + 1$ and $|X'| \leq k$, the amortized time complexity of this phase is $O(kd \log n)$.

Altogether, we have obtained an $O(7.66^k k^3 + 2^k k^3 d^3 \log n)$ -time dynamic graph algorithm for Feedback Vertex Set, where d is a degree bound on the graph.

8 Acknowledgement

Yoichi Iwata is supported by Grant-in-Aid for JSPS Fellows (256487). Keigo Oka is supported by JST, ERATO, Kawarabayashi Large Graph Project.

References

- [1] D. Berend and T. Tassa. Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 30(2):185–205, 2010.
- [2] A. Boral, M. Cygan, T. Kociumaka, and M. Pilipczuk. Fast branching algorithm for cluster vertex deletion. *CoRR*, abs/1306.3877, 2013.
- [3] Y. Cao, J. Chen, and Y. Liu. On feedback vertex set new measure and new structures. In *SWAT*, pages 93–104, 2010.
- [4] J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theor. Comput. Sci.*, 411(40-42):3736–3756, 2010.
- [5] C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the $o(n^2)$ barrier. In *FOCS*, pages 381–389, 2000.
- [6] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. In *STOC*, pages 159–166, 2003.
- [7] M. Doucha and J. Kratochvíl. Cluster vertex deletion: A parameterization between vertex cover and clique-width. In *MFCs*, pages 348–359, 2012.
- [8] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.
- [9] Z. Dvorak, M. Kupec, and V. Tuma. Dynamic data structure for tree-depth decomposition. *CoRR*, abs/1307.2863, 2013.
- [10] Z. Dvorak and V. Tuma. A dynamic data structure for counting subgraphs in sparse graphs. In *WADS*, pages 304–315, 2013.
- [11] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification-a technique for speeding up dynamic graph algorithms (extended abstract). In *FOCS*, pages 60–69, 1992.

- [12] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [13] M. R. Gary and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.
- [14] J. Guo, J. Gramm, F. Hüffner, R. Niedermeier, and S. Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *J. Comput. Syst. Sci.*, 72(8):1386–1396, 2006.
- [15] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *STOC*, pages 519–527, 1995.
- [16] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.
- [17] F. Hüffner, C. Komusiewicz, H. Moser, and R. Niedermeier. Fixed-parameter algorithms for cluster vertex deletion. *Theory Comput. Syst.*, 47(1):196–217, 2010.
- [18] G. F. Italiano, J. A. L. Poutré, and M. Rauch. Fully dynamic planarity testing in planar embedded graphs (extended abstract). In *ESA*, pages 212–223, 1993.
- [19] M. Patrascu and E. D. Demaine. Lower bounds for dynamic connectivity. In *STOC*, pages 546–553, 2004.
- [20] J. A. L. Poutré. Alpha-algorithms for incremental planarity testing (preliminary version). In *STOC*, pages 706–715, 1994.
- [21] F. Protti, M. D. da Silva, and J. L. Szwarcfiter. Applying modular decomposition to parameterized cluster editing problems. *Theory Comput. Syst.*, 44(1):91–104, 2009.
- [22] B. A. Reed, K. Smith, and A. Vetta. Finding odd cycle transversals. *Oper. Res. Lett.*, 32(4):299–301, 2004.
- [23] L. Roditty. A faster and simpler fully dynamic transitive closure. In *SODA*, pages 404–412, 2003.
- [24] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *FOCS*, pages 679–, 2002.
- [25] L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *FOCS*, pages 499–508, 2004.
- [26] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC*, pages 184–191, 2004.
- [27] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *ESA*, pages 580–591, 2004.
- [28] P. Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *FOCS*, pages 509–517, 2004.

- [29] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.
- [30] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *STOC*, pages 343–350, 2000.
- [31] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *SWAT*, pages 384–396, 2004.
- [32] M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *STOC*, pages 112–119, 2005.
- [33] C. Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *SODA*, pages 1757–1769, 2013.